# Dynamic Instruction Scheduling with a Scoreboard

Prof. Naga Kandasamy
ECE Department
Drexel University

April 27, 2016

These notes are derived from:

- J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, $5^{\text{th}}$ Edition, Morgan Kaufmann, 2011.

Dynamic scheduling is a technique whereby the hardware rearranges instruction execution to reduce stalls caused by data dependencies or data hazards. Dynamic scheduling offers several advantages: (i) it enables handling some cases when dependencies are unknown at compile time (e.g., because they may involve a memory reference); (ii) it simplifies the compiler; and (iii) it allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline.

Before describing a dynamic scheduling technique called scoreboarding, let us review the different types of data hazards in a typical pipeline.

## Data Hazards in a Pipeline

Data hazards may be classified as one of three types, depending on the order of read and write accesses of the instructions. By convention, the hazards are named by the sequential ordering in the program that must be preserved by the pipeline. Consider two instructions $i$ and $j$, with $i$ occurring before $j$. The possible data hazards are:

- **RAW (read after write):** Instruction $j$ tries to read a source before $i$ writes it, so $j$ incorrectly gets the old value. This is the most common type of hazard and the kind that we used forwarding to overcome in the MIPS pipeline. The following is an example of a RAW hazard:

  ```
  lw $1, 4($2)
  add $3, $1, $4
  ```

  A RAW dependency is called a "true dependency."

- **WAW (write after write):** Instruction $j$ tries to write an operand before it is written by $i$ and the writes end up being performed in the wrong order, leaving the value written by $i$ rather than the value written by $j$ in the destination. This hazard is present only in pipelines that

write in more than one pipe stage (or allow an instruction to proceed even when a previous instruction is stalled). Since the MIPS integer pipeline writes a register only in WB, it avoids this type of hazard. The following is an example of a WAW hazard:

```
add $1, $4, $2
sub $1, $5, $6
```

A WAW dependency is called an "output dependency."

- **WAR (write after read):** Instruction $j$ tries to write a destination before it is read by $i$, so $i$ incorrectly gets the new value. This cannot happen in the MIPS pipeline because all reads are early (in the ID stage) and all writes are late (in WB). This hazard occurs when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline. The following is an example of a WAR hazard:

  ```
  add $8, $1, $2
  sub $1, $5, $6
  ```

  A WAR dependency is called an "anti dependency."

Though WAW and WAR hazards do not happen in the MIPS pipeline due to its simple structure (in-order fetch, in-order execution, and in-order commit), we will shortly see how these hazards occur more easily when instructions are executed out of order. Also, the WAW and WAR hazards are also called "name dependencies" since the two instructions use the same register name or location but do not exchange any data between them.

# Dynamic Scheduling using a Scoreboard

A major limitation of the MIPS pipeline is that it uses in-order instruction issue. So, if an instruction stalls in the pipeline, no later instructions can proceed. Thus, if there is a dependence between two closely spaced instructions in the pipeline such as a `lw` and `add`, a stall will result. If there are multiple functional units, these units could lie idle. If instruction $j$ depends on a long-running instruction $i$, currently in execution in the pipeline, then all instructions after $j$ must be stalled until $i$ is finished and $j$ can execute. For example, consider this code:

```
divd $0, $2, $4
addd $10, $0, $8
subd $12, $8, $14
```

where `sudb`, `addd`, and `divd` are instructions that perform double precision subtract, add, and divide operations on 64-bit registers. Assume that `divd` takes 20 clock cycles to complete while `subd` and `addd`, being much simpler operations, each take 2 clock cycles to complete. The `subd` instruction cannot execute because the dependence of `addd` on `divd`, causing the pipeline to stall. However, `subd` is not data dependent on anything in the pipeline. This is a performance limitation that can be eliminated by not requiring instructions to execute in order.

In the MIPS pipeline, both structural and data hazards are checked during instruction decode (ID) and when an instruction could execute properly, it is issued from ID. To allow us to begin executing the `subd` in the above example, we must separate the issue process into two parts: checking the

structural hazards and waiting for the absence of a data hazard. We can still check for structural hazards when we issue the instruction; thus, we still use in-order instruction issue. However, we want the instructions to begin execution as soon as their data operands are available. Thus, the pipeline will do out-of-order execution, which also implies out-of-order completion.

By introducing the concept of out-of-order execution, we have split the ID stage into two:

- **Issue:** Decode instructions and check for any structural hazards.
- **Read operands:** Wait until there are no data hazards and then read operands.

An instruction fetch stage precedes the issue stage and may fetch instructions either into a single-entry latch or into a queue; instructions are then issued from the latch or queue. The EX stage follows the read operands stage, just as in the MIPS pipeline. If the pipeline supports floating-point operations, execution may take multiple cycles in EX, depending on the operation. Thus, we may need to distinguish when an instruction begins execution and when it completes execution; between the two times, the instruction is in execution. This allows multiple instructions to be in execution at the same time.

In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (that is, in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order. *Scoreboarding* is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependencies. It is named after the CDC 6600 scoreboard, which developed this technique.

Note that WAR and WAW hazards, which did not exist in the MIPS integer pipeline, can arise when instructions execute out of order. For example, consider the following code snippet:

```
divd $0, $2, $4
addd $10, $0, $8
subd $8, $8, $14
```

The `subd` destination is $8. Since `addd` is stalled waiting for `divd` to provide $0, it has not read the operand from $8 yet. This presents a WAR hazard between `addd` and `subd` since if the pipeline executes `subd` before `addd`—quite possible, since `divd` is a time-consuming operation—this will yield incorrect execution. Likewise, a WAW hazard would occur if the destination of `subd` were $10. Both these hazards are avoided in a scoreboard by stalling the later instruction involved in the hazards.

The scoreboard's goal is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible. Thus, when the next instruction to execute is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction. The scoreboard is responsible for instruction issue and execution, including all hazard detection. Taking advantage of out-of-order execution requires multiple instructions to be in their EX stage simultaneously which is achieved with multiple functional units.

In an out-of-order execution processor, scoreboards make sense primarily on the floating-point (FP) units since the latency of the other functional units (such as integer units) is very small. In our machine, shown in Fig. 1, assume that there are five functional units: two multipliers, one

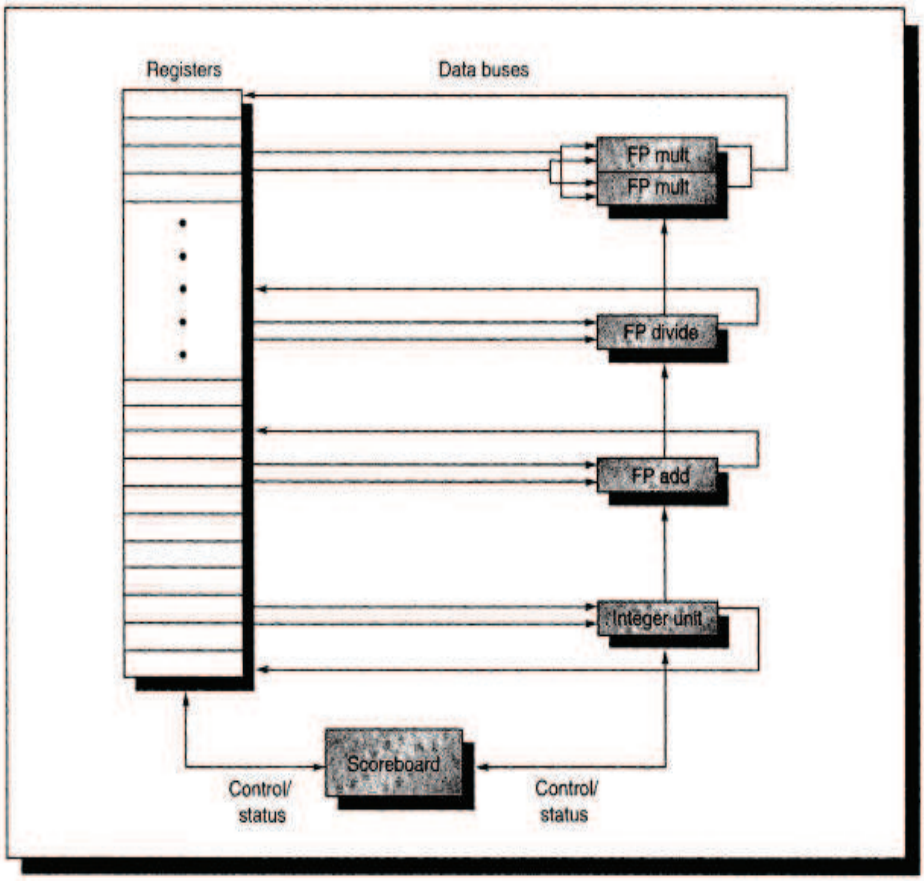


Figure 1: The basic structure of a multi-scalar processor with a scoreboard. The scoreboard controls instruction execution shown by the vertical control lines. The data flows between the register file and the functional units over buses shown by the horizontal lines. There are two FP multipliers, an FP divider, an FP adder, and an integer unit.

adder, one divide unit, and a single integer unit for all memory references, branches, and integer operations.

Every instruction goes through the scoreboard prior to execution, where a record of the data dependencies is constructed. This step corresponds to instruction issue and replaces part of the ID step in the MIPS pipeline. The scoreboard then determines when the instruction can read its operands and begin execution. If the scoreboard decides the instruction cannot execute immediately, it monitors every change in the hardware and decides when the instruction can execute. The scoreboard also controls when an instruction can write its result into the destination register. Thus, all hazard detection and resolution is centralized in the scoreboard.

Each instruction goes through four steps during the execution process. (Since we are focusing on FP operations, we will ignore the memory access step in the following discussion.) The four steps are as follows:

- **Issue:** If a functional unit required for the instruction is free and no other active instruction in the EX stage has the same destination register, the scoreboard issues the instruction to

the functional unit and updates its internal data structure. By ensuring that no other active functional unit wants to write its result into the destination register, we guarantee that WAW hazards cannot be present in the system. If a structural or WAW hazard is detected, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared. When the issue stage stalls, it will cause the buffer between instruction fetch and issue to fill; if the buffer is a single entry, instruction fetch stalls immediately. If the buffer is a queue with multiple instructions, it will stall when the queue fills up.

- **Read operands:** The scoreboard monitors the availability of the source operands. A source operand is available if no earlier issued active instruction is going to write it (a RAW hazard), or if the register containing the operand is being written by a currently active functional unit. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order.

  Because the operands for an instruction are read only when both operands are available in the register file, this scoreboard does not take advantage of forwarding. Instead registers are only read when they are both available.

- **Execution:** The functional unit begins execution upon receiving the source operands. When the result is ready, it notifies the scoreboard that it has completed execution. Different floating point operations incur different number of cycles.

- **Write result:** Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards and stalls the completing instruction, if necessary. A completing instruction is not allowed to write its results: (1) when there is an instruction that has not read its operands that precedes (i.e., in order of issue) the competing instruction and (2) one of the operands is the same register as the result of the completing instruction. If this hazard does not exist, or when it clears, the scoreboard tells the functional unit to store its result to the destination register. This step replaces the WB step in the simple MIPS pipeline.

Now let's look at a detailed example of how a scoreboard sequences the following instructions through the processor shown in Fig. 1:

```
ld F6, 34(R2)
ld F2, 45(R3)
multd F0, F2, F4
subd F8, F6, F2
divd F10, F0, F6
addd F6, F8, F2
```

Here the "F" registers store 64-but floating-point numbers. The scoreboard has three major components:

- **Instruction status:** Indicates which of the four steps the instruction is in.

- **Functional unit status:** Indicates the state of the functional unit (FU). There are nine fields for each functional unit:

Table 1: The status of instructions in the pipeline.

| Instruction status | | | | |
|---|---|---|---|---|
| Instruction | Issue | Read operands | Execution complete | Write result |
| `LD F6, 34(R2)` | ✓ | ✓ | ✓ | ✓ |
| `LD F2, 45(R3)` | ✓ | ✓ | ✓ | |
| `MULTD F0, F2, F4` | ✓ | | | |
| `SUBD F8, F6, F2` | ✓ | | | |
| `DIVD F10, F0, F6` | ✓ | | | |
| `ADDD F6, F8, F2` | | | | |

Table 2: The status of functional units in the pipeline.

| Functional unit status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Busy | Opcode | Fi | Fj | Fk | Qj | Qk | Rj | Rk |
| Integer | Yes | Load | F2 | R3 | | | | No | |
| Mult1 | Yes | Mult | F0 | F2 | F4 | Integer | | No | Yes |
| Mult2 | No | | | | | | | | |
| Add | Yes | Sub | F8 | F6 | F2 | | Integer | Yes | No |
| Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

Table 3: Register result status.

| Register result status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | F0 | F2 | F4 | F6 | F8 | F10 | F12 | . . . | F30 |
| Functional unit | Mult1 | Integer | | | Sub | Divide | | | |

  - Busy: Indicates whether the unit is busy or not.
  - Op: Operation to perform in the unit (e.g., add, subtract, or multiply).
  - Fi: Destination register number.
  - Fj and Fk: Source register numbers.
  - Qj and Qk: Functional units producing the values in the source registers Fj and Fk.
  - Rj and Rk: Flags indicating when Fj and Fk are ready.

- **Register result status:** Indicates which functional unit will write each register, if an active instruction has the register as its destination. This field is set to blank whenever there are no pending instructions that will write that register.

Tables 1 through 3 show the scoreboard's state after the `divd` instruction has issued. Each instruction that has issued or is pending issue has an entry in the instruction status table. There is one entry in the functional-unit status table for each functional unit. Once an instruction issues, the record of its operands is kept in the functional-unit status table. Finally, the register-result table indicates which unit will produce each pending result; the number of entries is equal to the number of registers. The instruction status register says that (1) the first `ld` has completed and written

Table 4: The status of instructions in the pipeline.

| Instruction status | | | | |
|---|---|---|---|---|
| Instruction | Issue | Read operands | Execution complete | Write result |
| `LD F6, 34(R2)` | ✓ | ✓ | ✓ | ✓ |
| `LD F2, 45(R3)` | ✓ | ✓ | ✓ | ✓ |
| `MULTD F0, F2, F4` | ✓ | ✓ | ✓ | |
| `SUBD F8, F6, F2` | ✓ | ✓ | ✓ | ✓ |
| `DIVD F10, F0, F6` | ✓ | | | |
| `ADDD F6, F8, F2` | ✓ | ✓ | ✓ | |

Table 5: The status of functional units in the pipeline.

| Functional unit status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Busy | Opcode | Fi | Fj | Fk | Qj | Qk | Rj | Rk |
| Integer | No | | | | | | | | |
| Mult1 | Yes | Mult | F0 | F2 | F4 | | | No | No |
| Mult2 | No | | | | | | | | |
| Add | Yes | Add | F6 | F8 | F2 | | | No | No |
| Divide | Yes | Div | F10 | F0 | F6 | Mult1 | | No | Yes |

Table 6: Register result status.

| Register result status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | F0 | F2 | F4 | F6 | F8 | F10 | F12 | . . . | F30 |
| Functional unit | Mult1 | | | Add | | Divide | | | |

its result, and (2) the second `ld` has completed execution but has not yet written its result. The `multd`, `subd`, and `divd` instructions have all issued but are stalled, waiting for their operands. The functional-unit status says that the first multiply unit is waiting for the integer unit, the add unit is waiting for the integer unit, and the divide unit is waiting for the first multiply unit. The `addd` instruction is stalled because of a structural hazard; it will clear when the `subd` completes. If an entry in one of these scoreboard tables is not being used, it is left blank. For example, the Rk field is not used on a load and the Mult2 unit is unused, and hence their fields have no meaning. Also, once an operand has been read, the Rj and Rk fields are set to No.

Now, assume the following EX cycle latencies for the floating-point functional units: Add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles. Tables 4 through 6 show the state of the scoreboard just before `multd` writes results to the register file. The `divd` has not yet read either of its operands, since it has a dependence on the result of the multiply. The $addd$ has read its operands and is in execution, although it was forced to wait until the `subd` finished to get the functional unit. Now, `addd` cannot proceed to write result because of the WAR hazard on register F6, which is used by `divd`.

Finally, Tables 7 through 9 show the scoreboard tables just before the `divd` goes to write result. `addd` was able to complete as soon as `divd` passed through read operands and got a copy of F6. Only the `divid` instruction remains to finish.

Table 7: The status of instructions in the pipeline.

| Instruction status | | | | |
|---|---|---|---|---|
| Instruction | Issue | Read operands | Execution complete | Write result |
| `LD F6, 34(R2)` | ✓ | ✓ | ✓ | ✓ |
| `LD F2, 45(R3)` | ✓ | ✓ | ✓ | ✓ |
| `MULTD F0, F2, F4` | ✓ | ✓ | ✓ | ✓ |
| `SUBD F8, F6, F2` | ✓ | ✓ | ✓ | ✓ |
| `DIVD F10, F0, F6` | ✓ | ✓ | ✓ | |
| `ADDD F6, F8, F2` | ✓ | ✓ | ✓ | ✓ |

Table 8: The status of functional units in the pipeline.

| Functional unit status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Name | Busy | Opcode | Fi | Fj | Fk | Qj | Qk | Rj | Rk |
| Integer | No | | | | | | | | |
| Mult1 | No | | | | | | | | |
| Mult2 | No | | | | | | | | |
| Add | No | | | | | | | | |
| Divide | Yes | Div | F10 | F0 | F6 | | | No | No |

Table 9: Register result status.

| Register result status | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | F0 | F2 | F4 | F6 | F8 | F10 | F12 | . . . | F30 |
| Functional unit | | | | | | Divide | | | |

To summarize, a scoreboard uses the available ILP to minimize the number of stalls arising from the program's true data dependencies. In eliminating stalls, a scoreboard is limited by: (1) The amount of parallelism available among the instructions which determines whether independent instructions can be found to execute. If each instruction depends on its predecessor, no dynamic scheduling scheme can reduce stalls; (2) The number of scoreboard entries which determines how far ahead the pipeline can look for independent instructions; and (3) The number and type of functional units that determines the importance of structural hazards.

The various scoreboard control algorithms are described below. `FU` stands for the functional unit used by the instruction, `D` is the name of the destination register, `S1` and `S2` are the source register names and `op` is the operation to be done. To access the scoreboard entry named `Fj` for the functional unit `FU`, we use the notation `Fj[FU]`. The result register field for register `D` is denoted by `Result[D]`.

```
function issue(op, D, S1, S2)
    // FU is the functional unit used by op.
    wait until (!Busy[FU] AND !Result[D]);
    Busy[FU] = Yes;
    Op[FU] = op;
    Fi[FU] = 'D'; // Register name is designated in quotes
```

```
    Fj[FU] = 'S1';
    Fk[FU] = 'S2';
    Qj[FU] = Result['S1'];
    Qk[FU] = Result['S2'];
    Rj[FU] = (Qj[FU] == 0);
    Rk[FU] = (Qk[FU] == 0);
    Result['D'] = FU;
```

The `read_operand` function waits until both source operand are available before proceeding. This takes care of the RAW hazard within the pipeline.

```
function read_operands(FU)
    wait until (Rj[FU] && Rk[FU]);
    Rj[FU] = No;
    Rk[FU] = No;

function execute(FU)
    // Execute whatever FU must do
```

The `write_back` function tests for the WAR hazard. This hazard exists if another instruction has this instruction's destination, `Fi[FU]`, as a source, either `Fj[f]` or `Fk[f]`, and if some other instruction has written the register, that is `Rj = Yes` or `Rk = Yes`.

```
function write_back(FU)
    wait until (forall f {(Fj[f] != Fi[FU] OR Rj[f] == No) AND
                          (Fk[f] != Fi[FU] OR Rk[f] == No)});
    foreach f do
        if Qj[f] == FU then Rj[f] = Yes;
        if Qk[f] == FU then Rk[f] = Yes;

    Result[Fi[FU]] = 0; // No FU generates the register's result
    Busy[FU] = No;
```